نام درس: سیستم های توزیع شده
نام و نام خانوادگی: محمد امانلو
شماره دانشجویی:
۸۱۰۱۰۰۰۸۴

تمرین شماره ۱

۱ مقدمه

در این گزارش قصد دارم نتایج حاصل از پروژه‌ی اول درس سیستم‌های توزیع‌شده را که شامل دو بخش است ارائه کنم. این پروژه با هدف آشنایی بیشتر با زبان برنامه‌نویسی Go و همچنین مفهوم مهم همروندی (Concurrency) و تست‌نویسی طراحی شده است. در بخش اول، یک سرور پایگاه داده‌ی Key-Value Database را به‌صورت همروند پیاده‌سازی کردم که از مکانیزم‌های goroutine و channel در زبان Go استفاده می‌کند. در بخش دوم، به نوشتن تست برای یک ماژول ساده به نام Squarer پرداختم که وظیفه‌اش دریافت اعداد و بازگرداندن مجذور آن‌ها است.
هدف اصلی از انجام این پروژه، درک بهتر مفاهیمی مانند همروندی Concurrency، تست‌نویسی صحیح در زبان Go و همچنین آشنایی با ابزارها و روال‌هایی است که توسعه‌دهندگان برای ارزیابی صحت و پایداری کدهای خود استفاده می‌کنند. در این گزارش، به بررسی مباحث همروندی این پروژه شامل دو بخش کاملاً جداگانه است:

- بخش اول: پیاده‌سازی Key-Value Server با استفاده از goroutine و channel در Go بدون استفاده از Mutex.
- بخش دوم: نوشتن تست برای یک برنامه‌ی Squarer که وظیفه‌ی مجذورگیری از اعداد دریافتی را برعهده دارد.

۲ شرح پروژه

در این بخش، جزییات پروژه را به تفصیل بیشتری توضیح می‌دهم. این پروژه دو هدف اساسی دارد:

۱. پیاده‌سازی یک سرور کلید-مقدار همروند با زبان Go.
۲. نوشتن تست‌هایی که رفتار ماژول Squarer را ارزیابی کند.

شرایطی که برای انجام پروژه تعیین شده، به شرح زیر است:

- استفاده از Locks و Mutexes ممنوع است.

- همگام‌سازی باید به‌طور کامل از طریق ابزارهای همروندی (goroutine، channel و select) در Go انجام شود.
- فقط از پکیج‌های استاندارد خاصی نظیر fmt، io، bytes، net، bufio و strconv می‌توان استفاده کرد.
- کد باید از استاندارد go fmt تبعیت کند.

۳ بخش اول: پیاده‌سازی سرور کلید-مقدار

در این بخش، من یک سرور Key-Value را به زبان Go نوشتم که می‌تواند همزمان به چندین کلاینت سرویس بدهد. این سرور چهار دستور اصلی دارد:

- Put: که برای اضافه کردن مقدار جدید به لیست پیام‌های مربوط به یک کلید استفاده می‌شود.
- Get: که مقادیر ذخیره شده را برمی‌گرداند.
- Delete: که تمامی مقادیر مرتبط با یک کلید خاص را حذف می‌کند.
- Update: که مقدار قبلی را با مقدار جدید جایگزین می‌کند و اگر آن مقدار قدیمی موجود نباشد، مقدار جدید را در انتهای لیست اضافه می‌کند.

برای جلوگیری از بلاک شدن سرور در حالتی که کلاینت‌ها کند باشند (Slow-Reading Clients)، یک کانال بافر شده‌ی ۵۰۰تایی برای ارسال پیام‌ها به هر کلاینت در نظر گرفته‌ام. اگر این کانال پر شود، پیام‌های جدید به‌صورت خودکار drop می‌شوند تا از مسدود شدن سرور جلوگیری شود.
سرور دارای دو شمارنده است:

- CountActive برای شمارش کلاینت‌هایی که در حال حاضر به سرور متصل هستند.
- CountDropped برای شمارش کلاینت‌هایی که از سرور قطع شده‌اند.

۱.۳ معماری کلی سرور

برای طراحی سرور، ابتدا در فایل server_impl.go ساختاری به‌نام keyValueServer تعریف کردم که شامل کانال‌ها، clients map و اطلاعات دیگری است که سرور برای اجرای عملیات‌ها نیاز دارد. مهم‌ترین اجزای این ساختار عبارت‌اند از:

- store: نمونه‌ای از واسط KVStore که در بخش دوم (kv_api.go و kv_impl.go) پیاده‌سازی شده است و عملیات Put، Get، Delete و Update روی پایگاه داده را انجام می‌دهد.

- newConnChan: یک channel برای اعلام اتصال جدید از طرف کلاینت‌ها به حلقه‌ی اصلی سرور.
- dropConnChan: یک channel برای اعلام قطع ارتباط کلاینت (توسط سرور یا به‌دلیل خطا/خروج).
- requestChan: کانالی که درخواست‌های مختلف را از تمام کلاینت‌ها دریافت می‌کند تا در حلقه‌ی runServerLoop پردازش شوند.
- closeChan و doneChan: کانال‌هایی برای مدیریت بسته‌شدن سرور و اتمام کار goroutineهای مرتبط.
- clients: یک map از شناسه‌ی کلاینت به ساختار client که به سرور متصل هستند. هر client کانال outChan مخصوص به خود را برای ارسال پیام از سرور به کلاینت دارد.

به منظور جلوگیری از Race Condition و بدون استفاده از Mutexes، از یک گوروتین اصلی (runServerLoop) استفاده کردم که تمامی دستورات را از کانال‌های مختلف می‌گیرد و پردازش می‌کند. تمام درخواست‌ها از سمت کلاینت‌ها ابتدا وارد یک کانال به نام requestChan می‌شوند و سپس در حلقه‌ی اصلی به‌ترتیب دریافت و اجرا می‌شوند. این روش، کاملاً جلوی وقوع Race Condition را می‌گیرد.
برای اتصال کلاینت‌ها، یک listener ساختم و در یک حلقه جداگانه (acceptLoop)، درخواست‌های اتصال را می‌پذیرم. هر کلاینتی که متصل شود، دو گوروتین جدید برای آن ساخته می‌شود:

- readLoop برای خواندن درخواست‌های ورودی از کلاینت
- writeLoop برای ارسال پاسخ‌ها به کلاینت

با این معماری، تنها حلقه‌ی اصلی سرور (runServerLoop) به store دسترسی مستقیم دارد و همه‌ی درخواست‌های Put، Get و غیره، ابتدا از طرف کلاینت‌ها در requestChan قرار می‌گیرند و سپس در همین حلقه‌ی اصلی پردازش می‌شوند. لذا، رقابت در دسترسی به store رخ نمی‌دهد و نیازی به Mutex نخواهد بود.

۲.۳ تست دستی با استفاده از telnet و netcat

برای تست کردن سرور به‌شکل دستی، من ابتدا فایل crunner.go را بررسی کردم که به‌صورت پیش‌فرض کامل نبود و صرفاً شامل یک پیام ساده "Not implemented." بود. بنابراین تصمیم گرفتم به‌جای تکمیل آن، از ابزارهای موجود مانند telnet یا netcat (nc) استفاده کنم.
برای نصب این ابزارها در محیط WSL (که در ادامه توضیح خواهم داد)، از دستورهای زیر استفاده کردم:

```
sudo apt update
sudo apt install telnet netcat
```

سپس با اجرای سرور روی پورت ۹۹۹۹ توانستم با این دستورات به سرور متصل شوم و به‌صورت دستی درخواست‌هایی را ارسال و نتایج را مشاهده کنم:

nc localhost 9999

Put:username:Hello World

Get:username

Update:username:Hello World:Goodbye

Delete:username

۳.۳ استفاده از WSL برای اجرای پروژه

از آنجا که فایل تست ارائه‌شده برای سرور، حاوی کدی به‌صورت زیر بود:

syscall.Syscall(syscall.SYS$_SETRLIMIT, syscall.RLIMIT_NOFILE, uintptr(64000), uintptr(64000))$

این خط از کد فقط در سیستم‌عامل لینوکس قابل اجرا بود و در محیط ویندوز عادی اجرا نمی‌شد. به همین دلیل، تصمیم گرفتم از WSL استفاده کنم. برای این کار در محیط Windows با استفاده از دستورات زیر WSL را نصب کردم:

wsl –install wsl –set-default-version 2

پس از نصب و راه‌اندازی، با ورود به محیط WSL و اجرای دستور go run و go test، کدها را تست کردم.
ابزارهایی مانند nc و telnet به من کمک کردند تا به‌صورت دستی تست‌هایی سریع و ساده انجام دهم و بتوانم رفتار سرور را در عمل ببینم.

۴.۳ مدیریت همزمانی با goroutine و channel

به‌محض اجرای متد Start، یک listener روی پورتی مشخص باز می‌شود (net.Listen)، سپس دو goroutine آغاز به کار می‌کنند:

۱. acceptLoop: در این حلقه، به‌طور مداوم متد Accept روی listener فراخوانی می‌شود تا اتصال‌های جدید شناسایی شوند. هر connection تازه از طریق کانال newConnChan به گوروتین اصلی (runServerLoop) اطلاع داده می‌شود.

۲. runServerLoop: این گوروتین با یک حلقه‌ی select دائماً چند کانال را گوش می‌دهد:

- newConnChan: وصل شدن کلاینت‌های جدید

- requestChan: دریافت درخواست‌های Put، Get، Delete، Update
- dropConnChan: قطع شدن کلاینت‌ها
- closeChan: بسته‌شدن سرور

۵.۳ رویه‌ی مدیریت یک کلاینت جدید

وقتی اتصالی در acceptLoop شناسایی می‌شود، شناسهٔ کلاینت تخصیص یافته و یک ساختار client ساخته می‌شود. این ساختار شامل موارد زیر است:

- conn: شیء net.Conn مربوط به ارتباط با کلاینت
- outChan: یک کانال ۵۰۰تایی که پیام‌های خروجی سرور برای این کلاینت را نگه می‌دارد

سپس دو goroutine برای هر کلاینت ایجاد می‌شود:

- readLoop: با استفاده از bufio.NewReader، پیام‌های ورودی کلاینت را سطر به سطر می‌خواند. هر خط دریافت شده parseLineToRequest و در requestChan قرار داده می‌شود.
- writeLoop: پیام‌هایی را که سرور می‌خواهد به کلاینت بفرستد در outChan قرار می‌گیرند می‌خواند و در conn می‌نویسد. اگر عملیات نوشتن با خطا مواجه شود، اتصال قطع شده و کلاینت drop خواهد شد.

۶.۳ نحوه‌ی پردازش درخواست‌ها (handleRequest)

در روش پیشنهادی، تمام متدهای کلیدی فقط از طریق حلقه‌ی اصلی سرور قابل فراخوانی هستند. به‌عنوان مثال:

- در فراخوانی Put، سرور مقدار newVal را به انتهای slice مربوط به آن key اضافه می‌کند.
- در Get، تمام مقادیر مربوط به key را بازیابی کرده و به‌صورت سطر به سطر برای کلاینت ارسال می‌کند. اگر صف ۵۰۰تایی پر باشد، آن پیام drop می‌شود تا سرور بلاک نشود.
- در Delete، تمامی مقادیر مرتبط با یک key حذف می‌شوند.
- در Update، اگر oldVal در لیست وجود داشت، جایگزین می‌شود؛ در غیر این صورت، newVal در انتهای لیست اضافه می‌گردد.

۷.۳ مدیریت disconnect

به‌محض رخ دادن خطا یا قطع شدن اتصال (مثلاً اتمام ورودی در readLoop یا بروز خطا در writeLoop)، شناسهٔ کلاینت روی dropConnChan قرار می‌گیرد. گوروتین اصلی با دریافت آن، کلاینت را از map حذف و کانال outChan آن را می‌بندد. همچنین، دو شمارنده با نام‌های activeCount و droppedCount برای گزارش تعداد کلاینت‌های فعال و تعداد کلاینت‌های قطع‌شده به‌روزرسانی می‌شوند.

۸.۳ CountActive و CountDropped

- CountActive: تعداد کلاینت‌هایی که در map سرور ثبت هستند و ارتباطشان هنوز قطع نشده است.
- CountDropped: مجموع کلاینت‌هایی که تاکنون به سرور وصل شده و سپس disconnect شده‌اند. هربار که کلاینتی قطع می‌شود، این شمارنده یک واحد افزایش می‌یابد.

۹.۳ Close و اتمام کار سرور

هنگامی که متد Close فراخوانی شود، کانال closeChan بسته می‌شود و حلقه‌ی اصلی با دریافت آن، اقدام به بسته‌شدن تمام کانکشن‌های جاری کرده، کانال‌های آن‌ها را می‌بندد و در نهایت listener.Close را فراخوانی می‌کند. این فرایند اطمینان حاصل می‌کند که هیچ goroutine اضافی در حال اجرا نماند.

۱۰.۳ جمع‌بندی بخش اول

در این بخش، سرور کلید-مقدار را به گونه‌ای طراحی کردم که کاملاً از مکانیزم‌های همروندی زبان Go استفاده کند و بدون استفاده از قفل، عملیات‌ها را با امنیت و کارایی مناسب انجام دهد. تست‌های دستی و خودکار نیز به من کمک کردند تا از صحت عملکرد کد اطمینان حاصل کنم. در ادامه، در بخش دوم به نوشتن تست‌های واحد برای ماژول دیگری به نام Squarer پرداختم. پیاده‌سازی بخش اول پروژه نشان می‌دهد که چگونه در زبان Go می‌توان یک سرور Key-Value را بدون قفل و صرفاً با اتکا بر goroutineها و channelها مدیریت کرد. این معماری اجازه‌ی Concurrency بالایی را می‌دهد و از بلاک شدن سرور در مواجهه با Slow-Reading Clients جلوگیری می‌کند. همچنین توابعی نظیر CountActive و CountDropped وضعیت اتصال کلاینت‌ها را برای ارزیابی و تست گزارش می‌دهند.

۴ بخش دوم: تست‌نویسی در زبان Go برای ماژول Squarer

در این بخش، وظیفه من نوشتن تست برای ماژولی به نام Squarer بود که اعداد صحیح را از طریق یک کانال ورودی دریافت می‌کند و سپس مجذور آن‌ها را در یک کانال خروجی برمی‌گرداند. هدف اصلی از این بخش، تمرین و تسلط بر مفاهیم تست‌نویسی در زبان Go و اطمینان از صحت پیاده‌سازی انجام‌شده بود.
ابتدا، کدهایی که از قبل در اختیار من قرار گرفت شامل فایل‌های زیر بود:

- squarer_api.go: در این فایل یک interface ساده با دو متد Initialize و Close تعریف شده بود. این متدها وظیفه‌ی Initialize و Close را بر عهده دارند. قرار بود که من این فایل را بدون تغییر نگه دارم و صرفاً از آن به عنوان راهنمایی برای نوشتن تست‌ها استفاده کنم.
- squarer_impl.go: در این فایل، یک پیاده‌سازی صحیح از interface فوق با نام SquarerImpl ارائه شده بود. این پیاده‌سازی، یک کانال ورودی برای گرفتن اعداد و یک کانال خروجی برای ارسال مربع آن‌ها داشت. یک گوروتین هم به‌طور مداوم از کانال ورودی اعداد را می‌خواند، آن‌ها را مربع می‌کرد و به کانال خروجی می‌فرستاد.
  یک نکته جالب این بود که در این پیاده‌سازی، کانال خروجی به صورت خودکار بسته نمی‌شد. این موضوع، باعث شد تست خاصی که من نوشتم (TestClosingInputChannel) در شرایط خاصی شکست بخورد که جلوتر آن را دقیق‌تر توضیح می‌دهم.

۱.۴ جزئیات تست‌های نوشته شده

من چهار تست مختلف در فایل squarer_test.go نوشتم. دلیل نوشتن هر تست و نحوه اجرای آن‌ها را به تفکیک توضیح می‌دهم:

۱.۱.۴ تست همروندی: TestConcurrentProcessing

در این تست چند عدد را همزمان به کانال ورودی ارسال کردم تا بررسی کنم آیا ماژول می‌تواند همزمان چند عدد را پردازش کند و به ترتیب مربع‌هایشان را برگرداند یا خیر. هدف این تست بررسی عملکرد ماژول در حالتی بود که اعداد به صورت سریع و پشت سر هم ارسال شوند. این تست را برای اطمینان از همروندی صحیح ماژول نوشتم.

۲.۱.۴ تست تعداد بالای داده‌ها: TestLargeInputSet

در این تست، من ۲۰۱ عدد متوالی را به کانال ورودی فرستادم و بررسی کردم که آیا تمام اعداد به درستی مجذور شده و در کانال خروجی ظاهر می‌شوند یا خیر. هدف از نوشتن این تست این بود که مطمئن شوم ماژول در حالت پردازش تعداد زیاد داده‌ها دچار مشکل نمی‌شود.

۳.۱.۴ تست بستن کانال ورودی: TestClosingInputChannel

یکی از مهم‌ترین تست‌هایی که نوشتم، همین تست بود. هدفم این بود که رفتار ماژول را در زمانی که کانال ورودی بسته می‌شود ارزیابی کنم. در این تست، ابتدا عدد -۸ را در کانال ورودی قرار دادم و بلافاصله بعد از آن کانال ورودی را بستم. سپس انتظار داشتم:

۱. مربع عدد -۸ از کانال خروجی خوانده شود.

۲. کانال خروجی (squares) نیز بسته شود و این موضوع را با دستور زیر چک کردم:

$$,ok :=< -squares; if ok\{t.Error(...)\}$$

اما این تست همیشه شکست می‌خورد. پس از بررسی دقیق، متوجه شدم که علت شکست تست این است که پیاده‌سازی داده شده در فایل squarer_impl.go هرگز کانال خروجی را نمی‌بندد و فقط کانال ورودی را مدیریت می‌کند. بنابراین، انتظار من در این تست نادرست بود و باعث شکست تست می‌شد. راه‌حل این بود که یا منطق تستم را تغییر دهم و از بستن کانال خروجی صرف‌نظر کنم یا اینکه پیاده‌سازی squarer_impl.go اصلاح شود که طبق دستورالعمل پروژه اجازه تغییر آن را نداشتم.

۲.۴ توضیح در مورد فایل crunner.go و تست دستی سرور

در ابتدای پروژه، یک فایل با نام crunner.go وجود داشت که صرفاً شامل یک پیغام "Not implemented" بود. این فایل قرار بود به من در اجرای دستی سرور کمک کند، اما چون کدی برای آن ننوشته بودند، به مشکل خوردم و صرفا از تست اتومات استفاده کردم.

۳.۴ ایجاد و مدیریت وابستگی‌ها با استفاده از go mod

یکی از چالش‌هایی که هنگام نوشتن برنامه‌ها در زبان Go داشتم، مدیریت وابستگی‌ها و کتابخانه‌های مختلفی بود که در کدم استفاده می‌شد. برای حل این چالش، از ابزاری به نام go modules استفاده کردم که روش استاندارد زبان Go برای مدیریت وابستگی‌ها و ورژن‌های آن‌ها است. در واقع، وقتی کدی می‌نویسیم که چندین بسته (package) مختلف را ایمپورت می‌کند، نیاز داریم به‌صورت مشخص معلوم باشد که این بسته‌ها دقیقاً از کجا باید دریافت شوند و نسخه دقیق هر کدام چه باشد. ابزار go modules برای من این مشکل را به‌خوبی حل کرد.
من برای این کار ابتدا در دایرکتوری myMainproject، دستور زیر را اجرا کردم:

```
go mod init myMainproject
```

با اجرای این دستور، فایلی با نام go.mod ساخته شد که به شکل زیر بود:

```
module myMainproject

go 1.24.1
```

در این فایل، کلمه module تعیین‌کننده‌ی نام منحصربه‌فردی است که برای پروژه خودم انتخاب کردم و باعث می‌شود تا وارد کردن و استفاده از بسته‌ها در داخل پروژه، به شکل منظمی انجام شود. همچنین، قسمت go 1.24.1 نسخه‌ای از کامپایلر زبان Go است که من در این پروژه استفاده کردم و تضمین می‌کند که کدها دقیقاً با این نسخه از زبان نوشته و اجرا شوند.

پس از ایجاد فایل go.mod، تمامی وابستگی‌هایی که لازم داشتم را به‌راحتی با استفاده از دستور زیر به پروژه اضافه کردم:

```
go mod tidy
```

این دستور، تمام بسته‌هایی که در پروژه من استفاده شده ولی هنوز دانلود نشده‌اند را شناسایی و دانلود می‌کند و همچنین هر بسته اضافی که دیگر در کد استفاده نمی‌شود را از لیست وابستگی‌ها حذف می‌کند. به این ترتیب، فایل go.mod من همیشه به‌روز می‌ماند.

بعد از ایجاد این فایل، نحوه‌ی import کردن بسته‌ها نیز در کدهای من ساده‌تر و مشخص‌تر شد. برای مثال، فرض کنید که پروژه من شامل یک دایرکتوری به نام p0partA است که در آن فایل‌هایی نظیر go.modserver$_i$$mpl.go$. و تعیین ماژول myMainproject، من به‌راحتی می‌توانم از داخل کدهایم این بسته را به صورت زیر ایمپورت کنم:

```
import "myMainproject/p0partA/kvstore"
```

با این روش، واضح است که هر کدام از فایل‌ها دقیقاً از کجا و به چه شکلی در پروژه وارد شده‌اند و این شفافیت کمک می‌کند که در پروژه‌های بزرگ‌تر، کدها به راحتی مدیریت شوند و احتمال بروز خطاهای وابستگی (dependency errors) به حداقل برسد.

استفاده از go mod همچنین باعث می‌شود که نسخه‌های مختلف وابستگی‌ها به شکل دقیقی مشخص باشند و در صورتی که نسخه‌ی جدیدی از یک بسته موجود باشد، بتوانم به آسانی آن‌ها را به‌روزرسانی کنم و از ایجاد ناسازگاری میان بسته‌ها جلوگیری کنم.

در نهایت، وقتی پروژه را روی یک کامپیوتر یا محیط دیگری منتقل کنم، کافی است فقط دستور go mod download را بزنم تا تمامی وابستگی‌های تعریف شده در فایل go.mod به صورت خودکار نصب شده و پروژه آماده‌ی اجرا شود:

```
go mod download
```

به این ترتیب توانستم ساختار منظم و قابل توسعه‌ای برای پروژه خود ایجاد کنم که در آن وابستگی‌ها به راحتی مدیریت می‌شوند و احتمال بروز خطا کاهش می‌یابد.

۴.۴ رعایت استانداردهای Go و استفاده از go fmt

در پایان پروژه، تمام کدم را با دستور زیر قالب‌بندی استاندارد کردم:

```
go fmt ./...
```

این دستور تمام فایل‌های Go را طبق استاندارد زبان Go قالب‌بندی می‌کند. من همچنین از قوانین نام‌گذاری زبان Go پیروی کردم، به‌عنوان مثال استفاده از نام‌های کوتاه و واضح، اسامی با حروف بزرگ برای متدهای exported و غیره، که همگی در مستندات رسمی Go آمده است.

۵ فایل‌های پروژه

۱.۵ فایل server_impl.go

پیاده‌سازی اصلی سرور کلید-مقدار در قالب یک ساختار همراه با goroutine و channel:

۲.۵ فایل server_test.go

حاوی تست‌های رسمی برای سرور کلید-مقدار. این فایل را نباید تغییر داد. بخشی از آن:

۳.۵ فایل kv_api.go و kv_impl.go

این دو فایل برای پیاده‌سازی اصلی پایگاه داده کلید-مقدار استفاده می‌شوند. kv_api.go یک interface به‌نام KVStore دارد و در kv_impl.go توابع Put، Get، Delete، Update پیاده‌سازی شده‌اند.

۴.۵ بخش دوم پروژه: فایل‌های squarer

squarer_api.go در این فایل یک interface به نام Squarer تعریف شده است که نباید آن را تغییر دهیم.

squarer_impl.go یک پیاده‌سازی درست (SquarerImpl) که تست‌ها باید آن را تأیید کنند:

squarer_test.go در این فایل، چندین تست برای بررسی رفتار SquarerImpl آمده است. در صورت اجرای این تست‌ها و موفقیت آنها، مشخص می‌شود پیاده‌سازی SquarerImpl درست است.

۶ مراجع استفاده‌شده

برای انجام این پروژه از منابع مختلفی استفاده کردم که هر کدام نقش مهمی در فهم بهتر زبان برنامه‌نویسی Go و رعایت استانداردهای آن داشته‌اند. در این قسمت منابعی را که برای نوشتن این گزارش و همچنین انجام پروژه به آن‌ها مراجعه کرده‌ام به همراه توضیح کوتاهی درباره‌ی کاربرد هر یک آورده‌ام:

- Effective Go - Formatting:
  از این منبع استفاده کردم تا با استانداردهای قالب‌بندی (Formatting) زبان Go آشنا شوم. همچنین از دستور go fmt که در این صفحه به آن اشاره شده بود استفاده کردم تا مطمئن شوم کدم استاندارد بوده و مطابق با الگوهای شناخته‌شده در زبان Go باشد.

- Effective Go - Names:
  این منبع را برای آشنایی با اصول نام‌گذاری صحیح در زبان Go مطالعه کردم. در طول انجام پروژه تلاش کردم تمام متغیرها، توابع و ساختارهای موجود در کدم از قواعد استاندارد نام‌گذاری پیروی کنند. برای مثال متغیرها و توابعی که عمومی (exported) بودند با حرف بزرگ و سایرین با حرف کوچک آغاز شده‌اند.

- Effective Go Documentation:
  این منبع کلی را نیز برای آشنایی بهتر با اصول برنامه‌نویسی به زبان Go مطالعه کردم و به کمک آن توانستم تسلط بیشتری بر نحوه‌ی کار کردن با مفاهیم اساسی مانند goroutine و channel کسب کنم.

- مستندات رسمی زبان برنامه‌نویسی Go (Go Official Documentation):
  برای فهم دقیق‌تر دستورات و توابع موجود در زبان Go و به خصوص نحوه‌ی نوشتن تست‌ها (go test) به این مستندات مراجعه کردم. این مستندات، توضیحات دقیقی درباره‌ی روش کار و مثال‌هایی کاربردی از دستورات مهم زبان ارائه کرده است.

- Using Go Modules (go mod):
  برای درک کامل مفهوم ماژول‌ها در زبان Go و اینکه چرا باید از go mod استفاده کنم، از این مستند استفاده کردم. از طریق این منبع متوجه شدم که استفاده از modules در مدیریت وابستگی‌ها و ورژن‌ها بسیار کمک‌کننده است و باعث می‌شود که کدم به‌راحتی در محیط‌های مختلف قابل اجرا و توسعه باشد.

- تور رسمی زبان Go (A Tour of Go):
  برای درک بهتر مکانیزم‌های همزمانی و ساختارهای ابتدایی مانند goroutine و channel به این منبع رجوع کردم. این تور ساده به من کمک کرد تا به‌راحتی درک عمیق‌تری نسبت به ساختارهای همروندی داشته باشم.